A developer's guide to the POWER architecture

POWER programming by the book

Level: Intermediate

Brett Olsson, Processor architect, IBM
Anthony Marsala (marsala@us.ibm.com), Software engineer, IBM

30 Mar 2004

POWER® processors are found in everything from supercomputers to game consoles and from servers to cell phones -- and they all share a common architecture. This introduction to the PowerPC application-level programming model will give you an overview of the instruction set, important registers, and other details necessary for developing reliable, high performing POWER applications and maintaining code compatibility among processors.

Show developerWorks content related to my search: powerpc registers

Show developerWorks source code related to my search: powerpc registers

The POWER architecture and the application-level programming model are common across all branches of the POWER architecture family tree. For detailed information, see the product user's manuals available in the IBM® POWER Web site technical library (see Resources for a link).

The POWER architecture is a Reduced Instruction Set Computer (RISC) architecture, with over two hundred defined instructions. POWER is RISC in that most instructions execute in a single cycle and typically perform a single operation (such as loading storage to a register, or storing a register to memory).

The POWER architecture is broken up into three levels, or "books." By segmenting the architecture in this way, code compatibility can be maintained across implementations while leaving room for implementations to choose levels of complexity for price/performances trade-offs. The levels are:

Book I. User instruction set architecture
: Defines the base set of user instructions and registers that should be common to all POWER implementations. These are the non-privileged instructions that are used by most programs.

Book II. Virtual environment architecture
: Defines additional user-level functionality that is outside the normal application software requirements, such as cache management, atomic operations, and user-level timer support. Although these operations are also non-privileged, a program would typically access the function via an operating system call.

Book III. Operating environment architecture
: Defines the privileged operations required and used at the operating system level. These include functions for memory management, exception vector processing, privileged register access, and privileged timer access. Direct hardware support for a variety of system services and capabilities would be specified in Book III.

Since the development of the original PowerPC architecture, deviations have focused on specific market segments. Today, there are two active branches of the PowerPC architecture family tree, *PowerPC AS* architecture and *PowerPC Book E* architecture. The PowerPC AS architecture is defined by IBM to meet the special needs of its eServer™ pSeries® UNIX and Linux server product family and its eServer™ iSeries® enterprise server product family. (See Resources for links to more information.) The PowerPC Book E architecture, referred to as Book E, is a collaboration between IBM and Motorola for the special

A natural history of POWER

POWER™ and PowerPC microprocessors have a long and storied past, starting with the IBM 801 and progressing in a none-too-linear fashion with the POWER, RS64, and PowerPC chip lines. Each chip family has brought its own set of strengths to requirements of the embedded market. Major differences from the original PowerPC architecture adopted in PowerPC AS and extensions adopted in Book E reside mostly in the area of Book III.

bear on the computing world, being implemented in everything from game consoles to mainframes, digital watches to high-end workstations. For the whole story, read "POWER to the people: A history of chipmaking at IBM."

There is also a modest set of application-level extensions in these derivative architectures, mostly related to application-specific opportunities, but PowerPC AS and PowerPC Book E share the base instruction set as defined in Book I of the PowerPC architecture. While the three architectures have differences that would primarily be exposed at the operating system level, they provide a great degree of application-level compatibility.

PowerPC originally defined support for both 32-bit and 64-bit implementations, with the ability to run 32-bit applications on a 64-bit system. PowerPC AS systems as used on the IBM pSeries and iSeries servers now offer 64-bit implementations of the architecture only, allowing both new 64-bit applications and legacy 32-bit applications to run on the same system. The PowerPC Book E architecture offers support for both 32-bit and 64-bit implementations, with 64-bit implementations also offering support for full 32-bit compatibility with PowerPC applications. Both architectures offer complete compatibility with PowerPC Book I instructions and registers, while offering system-level extensions in the area of memory management, exceptions and interrupts, timer support, and debug support.

The original PowerPC architecture remains an intact and integral part of both PowerPC AS and PowerPC Book E and presents a compelling story for application-level compatibility.

PowerPC application programming model

When working with more than one type of PowerPC processor, developers should keep in mind some differences in the way the processors handle memory.

PowerPC storage model

The PowerPC architecture has native support for byte (8-bit), halfword (16-bits), word (32-bit), and doubleword (64-bit) data types. PowerPC implementations can also handle string operations for multi-byte strings up to 128 bytes in length. 32-bit PowerPC implementations support a 4-gigabyte effective address space, while 64-bit PowerPC implementations support a 16-exabyte effective address space. All storage is byte addressable.

For misaligned data accesses, alignment support varies by product family, with some taking exceptions and others handling the access through one or more operations in hardware.

Big-endian or little-endian?

PowerPC, PowerPC AS, and the early IBM PowerPC 4xx family are primarily big-endian machines, meaning that for halfword, word, and doubleword accesses, the most-significant byte (MSB) is at the lowest address. Support for little-endian varies by implementation. PowerPC and PowerPC AS have minimal support, while the 4xx family provides more robust support for little-endian storage. Book E is endian-neutral, as the Book E architecture fully supports both accessing methods.

PowerPC application-level registers

PowerPC's application-level registers are broken into three classes: general-purpose registers (GPRs), floating-point registers (FPRs and Floating-Point Status and Control Register [FPSCR]), and special-purpose registers (SPRs). Let's look at each class.

General-purpose registers (GPRs)

The User Instruction Set architecture (Book I) specifies that all implementations have 32 GPRs (GPR0 - GPR31). GPRs are the source and destination of all integer operations and are the source for address operands for all load/store operations. They also provide access to SPRs. All GPRs are available for use with one exception: in certain instructions, GPR0 simply means the value `0`, and no lookup is done for GPR0's contents.

Floating-point registers (FPRs)

Book I specifies that all implementations have 32 FPRs (FPR0 - FPR31). FPRs are the source and destination operands of all floating-point operations and can contain 32-bit and 64-bit signed and unsigned integer values, as well as single-precision and double-precision floating-point values. They also provide access to the FPSCR.

Note that embedded microprocessors are frequently implemented without direct hardware support for the PowerPC floating-point instruction set, or only provide an interface to attach floating-point hardware. Many embedded applications have little or no need for floating-point arithmetic, and software emulation of PowerPC floating-point instruction execution is usually more than adequate when it is needed. The chip area and power savings of not implementing floating-point in hardware can be critical in embedded microprocessors.

The Floating-Point Status and Control Register (FPSCR) captures status and exceptions resulting from floating-point operations, and the FPSCR also provides control bits for enabling specific exception types, as well as for selecting one of the four rounding modes. Access to the FPSCR is through the FPRs.

Special-purpose registers (SPRs)

SPRs give status and control of resources within the processor core. SPRs that can be read and written by applications without support from a system service include the Count Register, the Link Register, and the Integer Exception Register. SPRs that can only be read by applications with support from a system service include the Time Base and other miscellaneous timers that may be supported.

- **Instruction Address Register (IAR)**
  This register is known to programmers as the *program counter* or *instruction pointer*. It is the address of the current instruction. This is really a pseudo-register, as it is not directly available to the user other than through a "branch and link" instruction. The IAR is primarily used by debuggers to show the next instruction to be executed.

- **Link Register (LR)**
  This register contains the address to return to at the end of a function call. Certain branch instructions can automatically load the LR to the instruction following the branch. Each branch instruction encoding has an LK bit. If the LK bit is 1, the branch instruction moves the program counter to the address in LR. Also, the conditional branch instruction `bclr` branches to the value in the LR.

- **Fixed-Point Exception Register (XER)**
  This register contains carry and overflow information from integer arithmetic operations. It also contains carry input to certain integer arithmetic operations and the number of bytes to transfer during load and store string instructions, `lswx` and `stswx`.

- **Count Register (CTR)**
  This register contains a loop counter that is decremented on certain branch operations. Also, the conditional branch instruction `bcctr` branches to the value in the CTR.

- **Condition Register (CR)**
  This register is grouped into eight fields, where each field is 4 bits. Many PowerPC instructions define bit 31 of the instruction encoding as the Rc bit, and some instructions imply an Rc value equal to 1. When Rc is equal to 1 for integer operations, the CR field 0 is set to reflect the result of the instruction's operation: Equal (EQ), Greater Than (GT), Less Than (LT), and Summary Overflow (SO). When Rc is equal to 1 for floating-point operations, the CR field 1 is set to reflect the state of the exception status bits in the FPSCR: FX, FEX, VX, and OX. Any CR field can be the target of an integer or floating-point comparison instruction. The CR field 0 is also set to reflect the result of a conditional store instruction (`stwcx` or `stdcx`). There is also a set of instructions that can manipulate a specific CR bit, a specific CR field, or the entire CR, usually to combine several conditions into a single bit for testing.

- **Processor Version Register (PVR)**
  The PVR is a 32-bit read-only register that identifies the version and revision level of the processor. Processor versions are assigned by the PowerPC architecture process. Revision levels are implementation defined. Access to the PVR is privileged, so that an application program can determine the processor version only with the help of an operating

system function.

PowerPC application-level instruction set

Table 1 lists different instruction categories and the types of instructions in each.

Table 1. Instruction categories

Instruction category	**Base instructions**
Branch	branch, branch conditional, branch to LR, branch to CTR
Condition register	crand, crnor, creqv, crxor, crandc, crorc, crnand, cror, CR move
Storage access	load GPR/FPR, store GPR/FPR
Integer arithmetic	add, subtract, negate, multiply, divide
Integer comparison	compare algebraic, compare algebraic immediate, compare logical, compare logical immediate
Integer logical	and, andc, nand, or, orc, nor, xor, eqv, sign extension, count leading zeros
Integer rotate/shift	rotate, rotate and mask, shift left, shift right
Floating-point arithmetic	add, subtract, negate, multiply, divide, square root, multiply-add, multiply-subtract, negative multiply-add, negative multiply-subtract
Floating-point comparison	compare ordered, compare unordered
Floating-point conversion	round to single, convert from/to integer word/doubleword
FPSCR management	move to/from FPSCR, set/clear FPSCR bit, copy FPSCR field to CR
Cache control	touch, zero, flush, store
Processor management	system call, move to/from special purpose registers, mtcrf, mfcr

Deciphering an instruction

All instruction encodings are 32 bits in length. Bit numbering for PowerPC is the opposite of most other definitions: bit 0 is the most significant bit, and bit 31 is the least significant bit. Instructions are first decoded by the upper 6 bits in a field, called the *primary opcode*. The remaining 26 bits contain fields for operand specifiers, immediate operands, and extended opcodes, and these may be reserved bits or fields. PowerPC defines the basic instruction formats listed in Table 2.

Table 2. PowerPC instruction formats

Format	**0**	**6**	**11**	**16**	**21**	**26**	**30**	**31**
D-form	opcd	tgt/src	src/tgt	immediate				
X-form	opcd	tgt/src	src/tgt	src	extended opcd			
A-form	opcd	tgt/src	src/tgt	src	src	extended opcd		Rc

BD-form	opcd	BO	BI	BD	AA	LK
I-form	opcd	LI			AA	LK

- **D-form**
  This instruction format provides up to two registers as source operands, one immediate source, and up to two registers as target operands. Some variations of this instruction format use portions of the target and source register operand specifiers as immediate fields or as extended opcodes.
- **X-form**
  This instruction format provides up to two registers as source operands and up to two target operands. Some variations of this instruction format use portions of the target and source operand specifiers as immediate fields or as extended opcodes.
- **A-form**
  This instruction format provides up to three registers as source operands, and one target operand. Some variations of this instruction format use portions of the target and source operand specifiers as immediate fields or as extended opcodes.
- **BD-form**
  This instruction format is used by the conditional branch instruction. The BO instruction field specifies the type of condition; the BI instruction field specifies which CR bit is to be used as the condition; and the BD field is used as the branch displacement. The AA bit specifies whether the branch is an absolute or relative branch. That is, the branch target address is the value of the immediate field or is the sum of the value of the immediate field and the address of the branch. The LK bit specifies whether the address of the next sequential instruction is saved in the Link Register as a return address for a subroutine call.
- **I-form**
  This instruction format is used by the unconditional branch instruction. Being unconditional, the BO and BI fields of the BD format are exchanged for additional branch displacement to form the LI instruction field. This instruction format also supports the AA and LK bits in the same fashion as the BD format.

As mentioned, there are variations to these instruction formats. However, these formats best represent the makeup of most of the PowerPC instruction set encodings.

Branch instructions

PowerPC provides a set of instructions for control flow that include:

- Both conditional and unconditional branch instructions
- A "decrement count and branch if zero/not zero" capability
- Absolute and relative branching
- Branch instructions using the contents of the Link Register or Count Register to specify the branch target address

The ability to save the address of the next sequential instruction is provided on all branch instructions, including the Branch to Link Register instruction. Conditional branches allow specifying any one of the 32 Condition Register bits to be used as the condition, and to specify whether that CR bit must be equal to 0 or 1 for the branch condition to succeed.

Condition register instructions

PowerPC provides a set of instructions for performing boolean operations on specific bits of the CR as well as copying CR fields. This allows the combining of multiple branching conditions, which can reduce the number of costly conditional branches. Table 3 lists PowerPC CR logical instructions.

Table 3. PowerPC CR logical instructions

Mnemonic	**Instruction name**
crand	CR logical and
crandc	CR logical and with complement
creqv	CR logical equivalent
crnand	CR logical not and

crnor	CR logical not or
cror	CR logical or
crorc	CR logical or with complement
crxor	CR logical xor

Integer arithmetic instructions

Many instructions exist for performing arithmetic operations, including add, subtract, negate, compare, multiply, and divide. Many forms exist for immediate values, overflow detection, and carry in and out. Multiply and divide instruction performance varies among implementations, as these are typically multi-cycle instructions. Table 4 lists PowerPC integer arithmetic instructions.

Table 4. PowerPC integer arithmetic instructions

Mnemonic	Instruction name
add[o][.]	add [& record OV] [& record CR0]
addc[o][.]	add carrying [& record OV] [& record CR0]
adde[o][.]	add extended [& record OV] [& record CR0]
addi	add immediate
addis	add immediate shifted
addic[.]	add immediate carrying [& record CR0]
addme[o][.]	add to minus one [& record OV] [& record CR0]
addze[o][.]	add to zero [& record OV] [& record CR0]
divd[o][.]	divide doubleword [& record OV] [& record CR0]
divdu[o][.]	divide doubleword unsigned [& record OV] [& record CR0]
divw[o][.]	divide word [& record OV] [& record CR0]
divwu[o][.]	divide word unsigned [& record OV] [& record CR0]
mulhd[.]	multiply high doubleword [& record CR0]
mulhdu[.]	multiply high doubleword unsigned [& record CR0]
mulhw[.]	multiply high word [& record CR0]
mulhwu[.]	multiply high word unsigned [& record CR0]
mulld[o][.]	multiply low doubleword [& record OV] [& record CR0]
mulli	multiply low immediate
mullw[o][.]	multiply low word [& record OV] [& record CR0]
neg[o][.]	negate [& record OV] [& record CR0]
subf[o][.]	subtract from [& record OV] [& record CR0]
subfc[o][.]	subtract from carrying [& record OV] [& record CR0]
subfe[o][.]	subtract from extended [& record OV] [& record CR0]

subfi	subtract from immediate
subfis	subtract from immediate shifted
subfic[.]	subtract from immediate carrying [& record CR0]
subfme[o][.]	subtract from to minus one [& record OV] [& record CR0]
subfze[o][.]	subtract from to zero [& record OV] [& record CR0]

Logical, rotate, and shift instructions

PowerPC provides a complete set of logical operations, and also provides support for sign-extension and counting the number of leading zeros in a GPR. Table 5 lists PowerPC logical instructions.

Table 5. PowerPC logical instructions

Mnemonic	Instruction name
and[.]	and [& record CR0]
andc[.]	and with complement [& record CR0]
andi.	and immediate & record CR0
andis.	and immediate shifted & record CR0
eqv[.]	equivalent [& record CR0]
nand[.]	not and [& record CR0]
nor[.]	not or [& record CR0]
or[.]	or [& record CR0]
orc[.]	or with complement [& record CR0]
oris	or immediate shifted
ori	or immediate
xor[.]	xor [& record CR0]
xoris	xor immediate shifted
xori	xor immediate
cntlzd[.]	count leading zeros doubleword [& record CR0]
cntlzw[.]	count leading zeros word [& record CR0]
extsb[.]	extend sign byte [& record CR0]
extsh[.]	extend sign halfword [& record CR0]
extsw[.]	extend sign word [& record CR0]

PowerPC provides a robust and powerful set of rotate and shift operations, as listed in Table 6.

Table 6. PowerPC rotate and shift instructions

Mnemonic	Instruction name
rldc[.]	rotate left doubleword then clear [& record CR0]
rldcl[.]	rotate left doubleword then clear left [& record CR0]

rldcr[.]	rotate left doubleword then clear right [& record CR0]
rldicl[.]	rotate left doubleword immediate then clear left [& record CR0]
rldicr[.]	rotate left doubleword immediate then clear right [& record CR0]
rldimi[.]	rotate left doubleword immediate then mask insert [& record CR0]
rlwimi[.]	rotate left word immediate then mask insert [& record CR0]
rlwinm[.]	rotate left word immediate then and with mask [& record CR0]
rlwnm[.]	rotate left word then and with mask [& record CR0]
sld[.]	shift left doubleword [& record CR0]
slw[.]	shift left word [& record CR0]
srad[.]	shift right doubleword [& record CR0]
sradi[.]	shift right doubleword immediate [& record CR0]
sraw[.]	shift right word [& record CR0]
srawi[.]	shift right word immediate [& record CR0]
srd[.]	shift right doubleword [& record CR0]
srw[.]	shift right word [& record CR0]

Floating-point instructions

PowerPC provides a robust set of floating-point arithmetic, comparison, and conversion operations. With software support, PowerPC floating-point arithmetic is fully compliant with the ANSI/IEEE Standard 754-1985 specification. Both single-precision and double-precision floating-point formats are supported in all arithmetic and comparison operations.

While floating-point data is stored in the FPRs in double-precision format, a set of single-precision arithmetic instructions perform the arithmetic operation and round the final result to single-precision while detecting exceptions (such as exponent overflow, underflow, and inexact) that should occur with a single-precision operation.

- A set of *Load Floating-point Single* instructions access the word in storage and convert that single-precision value to double-precision format before placing in the target FPR.
- A set of *Store Floating-point Single* instructions convert the source operand in the source FPR into single-precision format before storing to the target word in storage.

Specific floating-point exception classes can be enabled or disabled for supporting a trapping environment. Table 7 lists the base and optional PowerPC floating-point instruction set.

Table 7. PowerPC floating-point instructions

Mnemonic	Instruction name
fmr[.]	FP move [& record CR1]
fneg[.]	FP negate [& record CR1]
fabs[.]	FP absolute value [& record CR1]
fnabs[.]	FP negative absolute value [& record CR1]
fadd[s][.]	FP add [single] [& record CR1]

fsub[s][.]	FP subtract [single] [& record CR1]
fmul[s][.]	FP multiply [single] [& record CR1]
fdiv[s][.]	FP divide [single] [& record CR1]
fsqrt[s][.]	FP square root [single] [& record CR1]
fmadd[s][.]	FP multiply-add [single] [& record CR1]
fmsub[s][.]	FP multiply-subtract [single] [& record CR1]
fnmadd[s][.]	FP negative multiply-add [single] [& record CR1]
fnmsub[s][.]	FP negative multiply-subtract [single] [& record CR1]
fcmpo	FP compare ordered
fcmpu	FP compare unordered
fsel[.]	FP select [& record CR1]
frsp[.]	FP round to single [& record CR1]
fcfid[.]	FP convert from integer doubleword [& record CR1]
fctid[z][.]	FP convert to integer doubleword [& round to zero] [& record CR1]
fctiw[z][.]	FP convert to integer word [& round to zero] [& record CR1]
fres[.]	FP reciprocal estimate single [& record CR1]
frsqrte[.]	FP reciprocal square root estimate [& record CR1]

FPSCR manipulation instructions

Table 8 lists the base PowerPC FPSCR manipulation instruction set.

Table 8. PowerPC FPSCR manipulation instructions

Mnemonic	Instruction name
mcrfs	move to CR from FPSCR
mffs[.]	move from FPSCR
mtfsb0[.]	move to FPSCR bit 0
mtfsb1[.]	move to FPSCR bit 1
mtfsf[.]	move to FPSCR field
mtfsfi[.]	move to FPSCR field immediate

Load and store instructions

All load and store instructions are performed using the GPRs or a GPR and an immediate field in the instruction as specifiers of the address of the storage access. An option provided for most load and store instructions is to update the base register (in other words, RA) with the data's effective address generated by the instruction.

There are instructions for the following:

- Byte, halfword, word, and doubleword sizes
- Moving data between GPRs or FPRs and storage
- Moving data between GPRs or FPRs and storage

Special storage access instructions include:

- Multiple-word load/stores
  These are `lmw` and `stmw`, and can operate on up to 31 32-bit words.

- String instructions
  These can operate on up to 128-byte strings.

- Memory Synchronization instructions
  These are used to implement memory synchronization. Bit 2 (EQ bit) of the CR is set to record the successful completion of the store operation. Memsync instructions include:
  - `lwarx` (Load Word and Reserve Indexed)
  - `ldarx` (Load Doubleword and Reserve Indexed)
  - `stwcx` (Store Word Conditional Index)
  - `stdcx` (Store Doubleword Conditional Index)

  `lwarx`/`ldarx` performs a load and sets a reservation bit internal to the processor and hidden from the programming model. The associated store instruction `stwcx./stdcx.` performs a conditional store if the reservation bit is set and thereafter clears the reservation bit.

Stack

The PowerPC architecture has no notion of a stack for local storage. There are no push or pop instructions and no dedicated stack pointer register defined by the architecture. However, there is a software standard used for C/C++ programs called the Embedded Application Binary Interface (EABI), which defines register and memory conventions for a stack. The EABI reserves GPR1 for a stack pointer, GPR3-GPR7 for function argument passing, and GPR3 for function return values.

Assembly language programs wishing to interface to C/C++ code must follow the same standards to preserve the conventions.

Cache management instructions

The PowerPC architecture contains cache management instructions for both application-level cache accesses. Cache management instructions are listed in Table 9.

Table 9. Cache management instructions

Mnemonic	Instruction name
dcbf	Flush Data Cache Line
dcbst	Store Data Cache Line
dcbt	Touch Data Cache Line (for load)
dcbtst	Touch Data Cache Line (for store)
dcbz	Zero Data Cache Line
icbi	Invalidate Instruction Cache Line

Take care when porting cache manipulation code to a different PowerPC implementation. Although cache instructions may be common across different implementations, cache organization and size may likely change. For example, code that makes assumptions about the cache size to perform a flush may need to be modified for other cache sizes. Also, cache initialization may vary between implementations. Some provide hardware to automatically clear cache tags, while others require software looping to invalidate cache tags.

Self-modifying code

While writing self-modifying code is not a recommended practice, sometimes it is absolutely necessary. The following sequence shows the instructions used to perform a code modification:

1. Store the modified instruction.
2. Issue the `dcbst` instruction to force the cache line containing the modified instruction to storage.
3. Issue the `sync` instruction to ensure `dcbst` is completed.
4. Issue the `icbi` instruction to invalidate the instruction cache line that will contain the modified instruction.
5. Issue the `isync` instruction to clear the instruction pipeline of any instruction that may have already been fetched from

the cache line prior to the cache line being invalidated.
6. It is now okay to execute the modified instruction. An instruction cache miss will occur when fetching this instruction, resulting in the fetching of the modified instruction from storage.

Timers

Most implementations have provided a 64-bit timebase that is readable via two 32-bit registers or a single 64-bit register. The amount the timer increments varies across families, as do the SPR numbers and instructions to access the timebase. Therefore, take care when porting timer code across implementations. Additional timers may also vary, but most provide at least one kind of decrementing programmable timer.

Maintaining code compatibility

PowerPC users who expect to program for more than one implementation typically ask for tips on maintaining code compatibility. The following suggestions will help minimize porting problems:

- Use C code whenever possible.
  Today's C compilers can produce code that is comparable in performance to hand-assembly coding in many cases. C code, being Book I code, will guarantee code portability.

- Avoid processor-specific assembly instructions when possible.
  Try not to embed processor-specific assembly instructions in C, as they'll be harder to find. Isolate processor-specific code that is known to contain device-dependent registers or instructions. These are typically things like bootup sequences and device drivers, but also may include floating-point code (including long long types). Keep the assumptions and dependencies well documented.

- Use the Processor Version Register (PVR), but only when appropriate.
  Common code across minor variations of implementations is good, and the PVR can be used for decision making. But, in the case where major modifications are necessary (for example, PowerPC AS versus Book E MMU code), separate code bases are recommended.

Summary

Both PowerPC AS and PowerPC Book E support the application-level infrastructure defined in the original PowerPC architecture while providing optimizations for their specific target markets.

PowerPC AS is virtually identical to the original PowerPC in one of its two modes of operations, while PowerPC Book E has taken a different direction in its Book III-class definition, optimized for low-cost, low-power, yet architecturally flexible embedded implementations. Of course, doubleword integer instructions are not available on 32-bit implementations, and floating-point instructions are supported only through software emulation on most embedded implementations.

However, a significant opportunity is available for application binaries to move between the branches in the PowerPC architectural family tree.

Resources

- PowerPC segments architecture into three levels or "books" to maintain compatibility across implementations. Find all of the books and additional information in the PowerPC Architecture Book.

- For the complete history of POWER-related chipmaking at IBM, read "POWER to the people: A history of chipmaking at IBM" (*developerWorks*, March 2004).
- IBM's portal for the Power Architecture community is the first step in building a broader community around the Power Architecture. The portal provides a place to gather, find resources, and begin establishing a governance model to help guide future directions for innovation and collaboration.
- Find more information at the IBM PowerPC site.
- Detailed technical information is in user's manuals available at the IBM PowerPC Web site technical library. You will also find a wealth of information in the technical PowerPC Product Briefs and PPC white papers.
- PowerPC AS is virtually identical to the original PowerPC in one of its two modes of operation, while PowerPC Book E has optimizations for the embedded market.
- "PowerPC Assembly" (*developerWorks*, July 2002) presents an overview of assembly language from a PowerPC perspective and contrasts examples for three architectures.
- "A programmer's view of performance monitoring in the PowerPC microprocessor" (*IBM Systems Journal*, 1997) shows how you can analyze processor, software, and system attributes for a variety of workloads with the PowerPC's on-chip Performance Monitor (PM).
- The PowerPC Performance Libraries Project provides a number of optimized library functions for IBM PowerPC 4xx embedded processors. The libraries cover floating-point emulation and common C library string and memory functions.
- PMAPI is a library with a set of APIs that provide access to the hardware performance counters included in selected IBM POWERPC micro-processors.
- The PowerPC architecture is a Reduced Instruction Set Computer (RISC) architecture.
- The *developerWorks*Linux downloads and products page is an excellent starting point for software and literature for iSeries, pSeries, and other POWER-based platforms.
- To get started developing for Linux on POWER, read "EmPOWERing the Linux developer," a guide to the Linux distributions, compilers, and libraries you'll need to write enterprise applications. (*developerWorks*, March 2004).
- The goal of IBM's Linux on PowerPC team is to improve and extend Linux for embedded, 32-bit, and 64-bit processors. For compatibility information, documentation, code, and more, go to the Linux on PowerPC home page.
- IBM's Linux Technology Center provides a compilation of resources for POWER programmers.
- Linux on POWER gives an overview of POWER-related Linux offerings from IBM.
- Find more resources for Linux developers in the developerWorks Linux zone.
- Browse for books on these and other technical topics.

About the authors

Brett Olsson works on the VMX Architecture, PowerPC Book E Architecture team.

Anthony Marsala wears many hats as an actor, software engineer, author, husband, and most recently -- father. He has worked for the IBM embedded PowerPC group since 1993 where he helps create software tools for next-generation PowerPC processors. You can contact Anthony at marsala@us.ibm.com.

Share this....

Digg this story del.icio.us Slashdot it!